

CSC151.02 2014S, Class 21: Anonymous Procedures

Overview

- Preliminaries.
 - Upcoming Work
 - Admin.
 - Questions.
- Procedures as data.
- Lab.

Preliminaries

Upcoming Work

- Reading for Tuesday: Building Images by Iterating Over Positions
- Writeup: Exercise 5, Parts b-f
 - Subject: CSC 151.02 Writeup 15: Anonymous Procedures (OPTIONAL NAMES)
- Assignment 5.

Admin

- Our computer system is (still) acting up. Some aspects of lab may take longer than they should.
- A few of you are frustrated that your indentation changes when you cut and paste into Google docs. If you use Chromium for accessing Google docs, cut and paste *should* work.
- I *think* I only got one request for a partner on the current homework, but we'll do a quick survey.
- Study break tonight at 8pm.
- Extra credit:
 - Tuesday Town Hall, noon in JRC 101, 7:30 p.m. in Harris.
 - Tuesday wellness event.
 - Thursday CS Extras: New CS Curriculum.

Questions on Assignment 5

Procedures as Values

The computer system was acting up, so Sam wrote most of this on the whiteboard. This is Sam's attempt to record what we did.

We are considering procedures as values. What are the four questions we ask whenever we learn a new kind of value?

- How do we express the new kind of value?
- How does DrRacket express the value to us?
- What procedures do we use to work with the new kind of value?
- What kinds of things do we do with the new value?

How do we express procedures?

- `(lambda (PARAMS) EXPRESSIONS)`
- Use the name of an already-defined procedure.

How does DrRacket express the values to us?

- Usually with `#<procedure>`, or something similar

What procedures do we use to work with the new kind of value?

- `(map PROC LIST)` - build a new list by applying PROC to each value in LIST.
 - See the next section for our first explorations with `map`
- `(o PROC1 PROC2)` - build a new function that applies PROC2 then PROC1.
 - See the next section for our first uses of `compose`.
- `(l-s BINPROC VAL)` - build a new function that fills in the first parameter of a two-parameter procedure.
 - E.g., `(l-s + 2) => (lambda (_) (+ 2 _))`
 - E.g., `(l-s / 5) => (lambda (_) (/ 5 _))`
- `(r-s BINPROC VAL)` - build a new function that fills in the second parameter of a two-parameter procedure.
 - E.g., `(r-s + 2) => (lambda (_) (+ _ 2))`
 - E.g., `(r-s / 5) => (lambda (_) (/ _ 5))`

Some Examples

```
> (map square (list 1 2 3))  
'(1 4 9)
```

That is, apply `square` to 1, then to 2, then to 3, then put everything together into a list.

Note that the greater than sign is a prompt, which reminds us that we are working in the interactions pane.

Note that the single quotation mark tells us to treat the thing "verbatim" (as a list) rather than as a Scheme expression.

```
> (list 1 2 3)  
'(1 2 3)  
> '(list 1 2 3)  
'(list 1 2 3)
```

Now, let's explore the use of a lambda expression.

```
> (map (lambda (x) (+ 5 x)) (list 1 2 3))  
'(6 7 8)
```

Because $(+ 5 1)$ is 6, $(+ 5 2)$ is 7, and $(+ 5 3)$ is 8.

What happens if we put another expression in, as in the following?

```
> (map (lambda (x) (* 2 x) (+ 5 x)) (list 1 2 3))
```

- Hypothesis 1: It multiplies x by 2 and then multiplies by 5, so we should get $'(7 9 11)$.
- Hypothesis 2: The two operations are not combined so something else happens. But what else?
- Hypothesis 3: We get both values: $'(2 6 4 7 6 8)$
- Hypothesis 4: We get two lists.
- The answer: Scheme always uses the value of the *last* expression in the body of a procedure, so we just get the $(6 7 8)$

```
(map (lambda (x) (* 2 x) (+ 5 x)) (list 1 2 3)) '(6 7 8)
```

Okay, now let's consider what happens with a slight variant.

```
> (map (lambda (x) (display x) (+ 5 x)))
```

In this case, `display` has a side effect. Hence, we'll expect to see some output in addition to the sum. It turns out that `map` does not guarantee the order in which we visit the elements in the list. Hence, we might see 123 before the list, or we might see 213 or ...

We can now turn our attention to function composition. Recall that $(\circ f g)$ creates a new function that applies g and then f . One simple example:

```
> (map (o square square) (list 1 2 3))  
'(1 16 81)
```

Why? For the first element, we square 1 and then square it again, which is still 1. For the second element, we square 2, yielding 4, and then square that, yielding 16. For the third element, we square 3, yielding 9, and then square it again, yielding 81.

But that example uses the same function twice. What if we use different functions?

```
> (map (o square increment) (iota 4))  
'(1 4 9 16)
```

Let's see $(iota 4)$ is the list $'(0 1 2 3)$. For each element, we increment the element and then square it. 0 plus 1 is 1, 1 squared is 1. Our first element is 1. 1 plus 1 is 2. 2 squared is 4. Our next element is 4. And so on and so forth.

Does it make a difference if we reverse the order?

```
> (map (o increment square) (iota 4))  
'(1 2 5 10)
```

Once again, `(iota 4)` is the list `(0 1 2 3)`. We square 0, yielding 0, and then add 1, yielding 1. For the next element, we square 1, yielding 1, and then add 1, yielding 2. For the next element, we square 2, yielding 4, and then add 1, yielding 5. For the last element, we square 3, yielding 9, and then add 1, yielding 10.

Anyway, it's clear that the order in which we compose functions makes a difference.

Now, let's turn our attention to sectioning.

```
> (map (l-s * 5) (iota 4))  
'(0 5 10 15)
```

The `(l-s * 5)` is "multiply 5 by x". So, we multiply 5 by 0, 5 by 1, 5 by 2, and 5 by 3. Does it make a difference if we use right-sectioning?

```
> (map (r-s * 3) (iota 4))  
'(0 3 6 9)
```

The `(r-s * 3)` is "multiply x by 3". So, we multiply 0 by 3, 1 by 3, 2 by 3, and 3 by 3. Yes, there's a subtle difference in how we expressed these. It clearly didn't matter for multiplication, but it may matter for other operations.

```
> (map (r-s - 2) (iota 4))  
'(-2 -1 0 1)
```

The `(r-s - 2)` is "subtract 2 from x". So we subtract 2 from 0, yielding -2. We subtract 2 from 1, yielding -1. We subtract 2 from 2, yielding 0. And we subtract 3 from 2, yielding 1.

```
> (map (l-s - 2) (iota 4))  
'(2 1 0 -1)
```

The `(l-s - 2)` is "subtract x from 2". So we subtract 0 from 2, yielding 2. We subtract 1 from 2, yielding 1. We subtract 2 from 2, yielding 0. And we subtract 3 from 2, yielding -1.

Why do we need the sectioning and the map? We can't we must make iota a parameter to the subtraction operation?

```
> (- 2 (iota 4))
```

Well, subtraction assumes that both of its parameters are numbers.

`(iota 4)`, in contrast, is a list. We need `map` to do something for each element of the list. And we need a way to write "subtract from 2" to distinguish it from "negative 2".

Lab

Reflection

Samuel A. Rebelsky, rebelsky@grinnell.edu

Copyright (c) 2007-2014 Janet Davis, Samuel A. Rebelsky, and Jerod Weinman. (Selected materials are copyright by John David Stone or Henry Walker and are used with permission.)



This work is licensed under a Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.